

# DYNAMIC LOAD-BALANCING FOR A PARALLEL ELECTROMAGNETIC PARTICLE-IN-CELL CODE\*

David B. Seidel, Steven J. Plimpton, Michael F. Pasik, Rebecca S. Coats

Sandia National Laboratories  
Albuquerque, NM 87185-1152

Gary R. Montry

Southwest Software  
Albuquerque, NM 87111

## Abstract

QUICKSILVER is a 3-D electromagnetic particle-in-cell simulation code developed and used at Sandia to model relativistic charged particle transport [1]. It was originally written for shared-memory, multi-processor supercomputers such as the Cray X/MP.

A new parallel version of QUICKSILVER has been developed [2] to enable large-scale simulations to be efficiently run on massively-parallel distributed memory supercomputers with thousands of processors, such as the DOE ASCI (Accelerated Strategic Computing Initiative) platforms. The new parallel code implements all features of the original QUICKSILVER and runs on any platform that supports the message-passing interface (MPI) standard [3] as well as on single-processor workstations.

The original QUICKSILVER code was based on a multiple-block grid, which provided a natural strategy for extending the code to partition a simulation among multiple processors. By adding the automated capability to divide QUICKSILVER's existing blocks into sub-blocks and then distribute those sub-blocks among processors, a simulation's spatial domain can be easily and efficiently partitioned. Based upon this partitioning scheme as well as QUICKSILVER's existing particle-handling infrastructure, an efficient algorithm has been developed for dynamically rebalancing the particle workload on a timestep-by-timestep.

This paper will elaborate on the strategies used and describe the algorithms developed to parallelize and dynamically load-balance the code. Results of several benchmark simulations will be presented that illustrate the code's performance and parallel efficiency for a wide variety of simulation conditions. These calculations have as many as  $10^8$  grid cells and  $10^9$  particles and were run on thousands of processors.

## I. Field Solver Parallelization

To decompose a simulation's finite-difference grid over multiple processors, QUICKSILVER's preprocessor, MERCURY, was modified to subdivide the user-supplied, possibly multi-block, grid into many sub-blocks and

distribute those blocks among the processors. MERCURY provides detailed user control over the way in which the original blocks are subdivided and how they are then distributed to processors (in general, multiple sub-blocks/processor). By default the algorithm will, as best it can, decompose using one sub-block per processor, with all blocks being the same (roughly cubical) size. This will balance the field computation, while minimizing inter-processor communication. Fig. 1 shows a 2-D example of this strategy for an eleven-block decomposition. Initially, a single cut A is made, then two cuts B, etc.

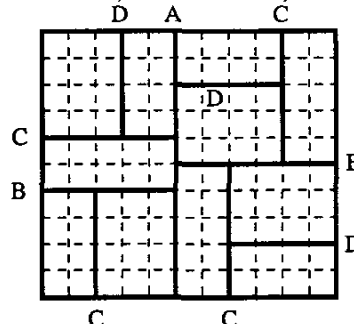


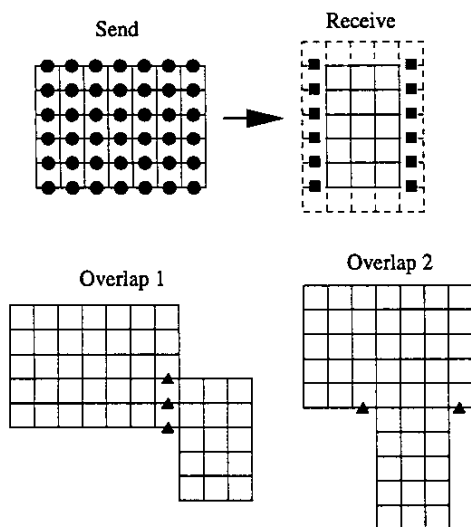
Figure 1. A 2-D schematic of a single block decomposed into eleven blocks.

Field "connections" between adjoining blocks take place at two points during a timestep. The first is after **E** and **B** fields have been updated in each block. The second is after **J** and **p** fields have been created by scattering particle current and charge. In both cases, fields have been computed within individual blocks. Before the timestep can proceed, field values at or near block boundaries must be exchanged between blocks.

A convenient paradigm for the communication required for this connection is to define a "send" set of all values of a given block where the computed values of a specific field component are known to be valid, and a "receive" set of all values of another block where needed values of that component are known to be invalid (or incomplete in the case of **J** or **p**). An "overlap" set can be defined to be the intersection of these two sets and represents the data that must be sent from the "send" block to the "receive" block. Fig. 2 shows a 2-D example of such a connection for the  $E_x$  field component. Each processor constructs a

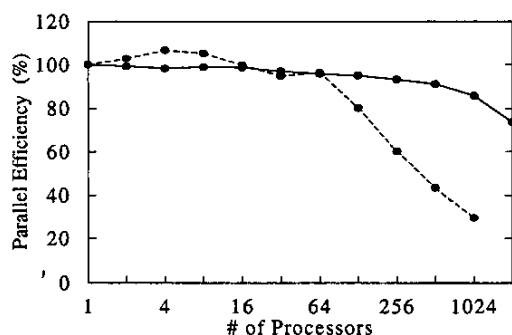
\* Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

plan that describes the overlap sets it needs to exchange (send and receive) with other processors. The functions that create and use these fairly complex data structures are written in C. For a more detailed description, see [2].



**Figure 2.** A 2-D diagram of edge-centered  $E_x$  field components in two blocks. Above are the “send” and “receive” sets; below two possible overlap sets are shown for two different ways the blocks adjoin.

To assess the performance two sets of simulations were performed on Sandia’s Intel Tflops machine. The first was a single  $80 \times 100 \times 96$  grid block (768,000 cells) run with typical boundary conditions, repeated for several different numbers of processors. The second problem was similar to the first, except that the number of grid cells was scaled with the number of processors (27,000 cells/processor). We compute parallel efficiency by dividing the 1-processor run time by the product of  $P$  and the  $P$ -processor run time, where  $P$  is the number of processors. The results are summarized in Fig 3.



**Figure 3.** Parallel efficiency for both a fixed-size (dashed) and scaled-size (solid) problem.

The data shows that the algorithm is very efficient as long as each processor has a sufficient number of cells; as the number of cells per processor goes down, the ratio of

communication (overhead) to computation (work) goes up. We believe that the observed super-linear performance (efficiencies greater than 100%) is due to cache effects. When the problem size per processor is reduced enough that significant portions of the field arrays fit in cache, the field update computations speed up.

## II. Parallelization of Particle Handling

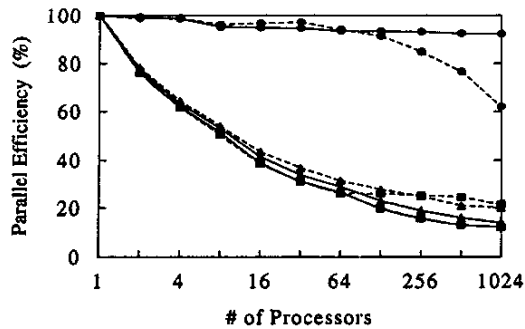
In the original version of QUICKSILVER, particle state information (position, momentum, charge) is stored in fixed-length containers referred to as caches. Caches are dynamically allocated as needed and can be collected in lists that represent the state of all simulation particles at any given time. Since particles are continuously entering and leaving the simulation, the code maintains an input cache list, describing the particles at the beginning of the timestep. As particles from this list are advanced in time, those surviving, as well as any created, are placed in caches in the output cache list. Then at the end of the timestep, the input caches are discarded, and the output list becomes the input list for the next timestep.

With this infrastructure in place, it was straightforward to extend the code for distributed memory operation by adding a third cache list (the migrate list) and adding a few extra steps to the particle-handling process. Now, after a particle has been advanced in time, in addition to determining whether or not it survives, we also must determine if it has moved into a block owned by another processor, in which case it is placed in a cache on the migrate list. Since local field values at each particle’s location are needed to advance the particle, and each particle needs access to the current density and charge fields in order to allocate contributions due to their motion and position, it is desirable to assign particles to the processor that owns the block in which they are located. Hence all particles in the migrate cache list will need to be transferred to the processor owning their new block.

After all of the particles on all processors have been advanced, each processor counts how many particles in its migrate caches need to be sent to each of its neighbor processors (processors that own a block that adjoins one of its blocks) and sends that count to each neighbor. After this information is communicated, each processor can post a receive, with adequate buffer space for the message, for each processor that will be sending it migrating particles. Now each processor bundles the particle state data destined for each neighbor processor and sends a message containing that data. Finally, it waits for the messages for which it earlier posted receives, and as those messages arrive, it puts the particles contained in each message into caches on its own output cache list.

To assess the performance of the particle handler, six series of problems were performed. The first two had a spatially uniform particle density (all particles were moving at  $\sim 1/2$  cell per timestep, but in such a way that the density remained uniform). One series was a fixed-size problem with  $\sim 262K$  cells and 3.15M particles. The

other was scaled with 27K cells and 324K particles per processor. The performance is shown in Fig. 4, which indicates that the efficiency remains quite high until the number of particles per processor falls below ~30K per processor, at which point the communications overhead starts to become significant.



**Figure 4.** Parallel efficiencies for particle handling. Solid and dashed lines are efficiencies for scaled-size and fixed-size problems, respectively. Circles indicate uniform particle density; squares and triangles indicate static and dynamic imbalanced cases, respectively.

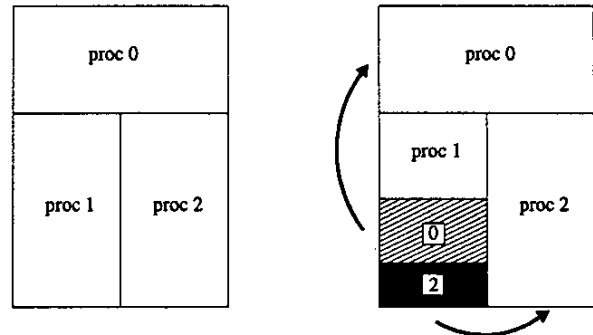
We also performed benchmarks with both static and dynamic imbalanced particle distributions, with both fixed and scaled problem size. The fixed-size problem is the same as the uniform case previously described except that ~2% of the cells contained four times the average number of particles per cell, 14% had 3.7 times the average, 42% had 1.3 times the average, and 42% contained no particles at all. The scaled-size problem was also the same as the uniform case but the imbalance was increased such that 0.1% of the cells contain ten times the average number of particles and ~73% have no particles. The particles were loaded in such a way that although moving at high velocity, their distribution remained unchanged in the static case and moved rapidly along the diagonal of the simulation extent in the dynamic case. Fig. 4 shows the performance for these two cases. Note that the fixed-size problem can never have a processor with more than four times the average workload and this limit is reached for 64 or more processors. Similarly, the scaled-size problem is limited to ten times the average workload, which occurs for 1000 or more processors. The performance curves in Fig. 4 reflect this loss of efficiency due to imbalance in the particle workload.

### III. Load Balancing

There are several approaches to deal with workload imbalance caused by non-uniform particle distribution. The algorithm we implemented, which will be described here, was chosen for three primary reasons. First, it is reasonably efficient with regard to the amount of overhead required for inter-processor communication. Second, the cost and complexity associated with dynamically rebalancing as the simulation's particle

distribution changes during the course of the simulation is reasonably low. Finally, the algorithm requires surprisingly little modification to the existing particle handling described in the previous section.

The concept we have adopted continues to use the static decomposition of the field grids, but dynamically migrates particles from overworked processors to underworked ones via "windows" within a processor's blocks. A "window" is a contiguous sub-region of grid cells within a heavily loaded block that is mapped to a new block on a lightly loaded processor, as illustrated in Fig. 5.



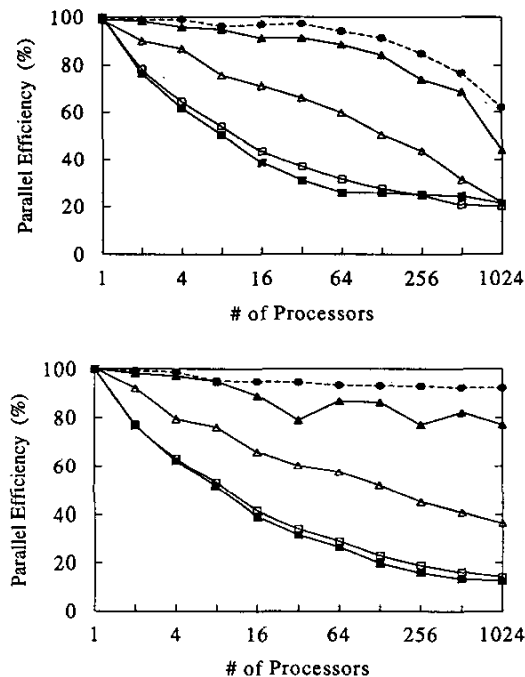
**Figure 5.** A three-processor decomposition with one block assigned to each processor (on left). Shaded regions of processor 1's heavily loaded block are designated as "windows" and assigned to processors 0 and 2 (on right).

Particles within the window migrate from the "parent" block (on the heavily loaded processor) to a new "child" block (on the lightly loaded processor). The child processor pushes those particles as long as they remain inside the window region. Particles that enter/exit the window migrate between the parent and child processors. For example, on the left side of Fig. 5 each of 3 processors initially owns one block. If processor 1's block (the parent) has too many particles, two (shaded) window regions (the children) are created, one each for processors 0 and 2. Processors 0 and 2 will each push particles in two blocks, their original block and their new child block.

Note that within the window regions, the child processor will only push particles; the parent processor will continue to compute **E** and **B** field updates in these regions in order to maintain load balance in the field computations. Since the child's processors new particles actually reside (in a geometric sense) in the parent block, communication of additional field information between the parent and child processors is required. In practice achieving load balance in both the particle push and the field update more than compensates for this extra overhead. QUICKSILVER maintains this balance by dynamically adjusting the number and sizes of windows based upon threshold parameters supplied by the user.

To assess the performance of the load-balancing algorithm, the four load-imbalanced simulation series described in the previous section were repeated with the load balancer turned on. The results are shown in Fig. 6. It is seen that for the case of static imbalance, the algorithm

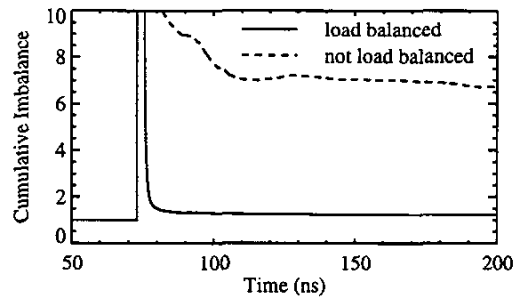
works quite well, almost equaling the performance of the case of uniform particle distribution and a significant improvement over not using the load balancer. On the other hand, for the case of dynamic imbalance the performance is significantly reduced, although it remains a clear improvement over the unbalanced case. It should be noted that the dynamic benchmark is particularly stressful to the algorithm because all the particles are moving in a single direction at high velocity and consequently rebalancing is frequently required (one in four timesteps for 512 processors). Since every time a rebalance occurs, particles are pushed for one step in a totally unbalanced manner, this represents a significant (and probably unrealistic for most real problems) loss.



**Figure 6.** Parallel efficiencies for fixed-size (upper) and scaled-size (lower) problems. Squares and triangles show load balancing turned off and on, respectively. Shaded and open symbols indicate static and dynamic imbalance in particles, respectively. The dashed lines are for the corresponding uniform-load simulations.

To assess the performance of the load balancer for a realistic application, we present data collected by T. D. Pointon [4] using this algorithm to simulate the vacuum power flow section of Sandia's Z accelerator. The simulation was performed on Sandia's Tflop computer using 100 processors. The simulation ran for 500K timesteps with 780K cells and 1.2M particles (average) with a spatially and temporally non-uniform distribution. We can define a measure of the imbalance to be the ratio of the maximum particle count on any processor and the average particle count over all processors. A value of one indicates perfect balance; higher values represent the increase in run time over the perfectly balanced case since

the run time of the processor with the most particles determines the time required by the simulation. Using this single-timestep value, we can then compute a cumulative imbalance by taking its particle-weighted average over time. This cumulative imbalance is plotted as a function of time in Fig. 7 for the simulation with and without load balancing. By the end of the simulation, the algorithm has improved the imbalance from  $\sim 6.7$  to  $\sim 1.25$ . On average, the algorithm rebalanced once every 400 timesteps and one cell in every seven was in a window block.



**Figure 7.** Cumulative load imbalance for a 100-processor simulation of the electron flow in the Z accelerator.

## IV. Conclusions

A new parallel version of QUICKSILVER has been developed that scales well to large numbers of distributed memory processors. A novel algorithm using "window blocks" provides efficient load balancing of the particle computation, even for highly non-uniform (spatial and/or temporal) particle distributions. Setup and domain decomposition are handled by the MERCURY pre-processor and QUICKSILVER's original wide variety of boundary conditions, sources, and output diagnostics are still available.

## V. References

- [1] D. B. Seidel, M. L. Kiefer, R. S. Coats, T. D. Pointon, J. P. Quintenz, and W. A. Johnson, "The 3-D, Electromagnetic, Particle-In-Cell Code, QUICKSILVER," in *The CP90 Europhysics Conf. on Computational Physics*, A. Tenner, Ed., World Scientific, Singapore.
- [2] S. J. Plimpton, D. B. Seidel, M. F. Pasik, and R. S. Coats, "Load-Balancing Techniques for a Parallel Electromagnetic Particle-in-Cell Code," Sandia Natl. Laboratories, Albuquerque, NM, Tech. Rep. SAND2000-0183, Jan. 2000.
- [3] Argonne National Laboratories, "The Message Passing Interface (MPI) Standard," Available: <http://www-unix.mcs.anl.gov> Directory: mpi File: index.html.
- [4] T. D. Pointon and W. A. Stygar, "3-D Simulations of the Electron Flow in the Vacuum Power Flow Section of the Z Accelerator," P4-E08, this proceedings.